Hybrid Apps Using C# and JavaScript with ChakraCore

June 2, 2016

ChakraCore provides a high performance JavaScript engine that powers the Microsoft Edge browser and Windows applications written with WinJS.

There are many reasons why one may want to embed JavaScript capabilities into an app. One example may be to take a dependency on a JavaScript library that has not yet been ported to the language you're developing in. Another may be that you want to allow users to "eval" small routines or functions in JavaScript, e.g., in data processing applications. The key reason for our investigation of ChakraCore was to support the React Native framework on the Universal Windows Platform, which is a framework for declaring applications using JavaScript and the React programming model.

Hello, ChakraCore

Embedding ChakraCore in a C# application is quite easy. To start, the grab a copy of the JavaScript runtime wrapper from GitHub. All examples throughout this document will assume this wrapper is being used. Include this code directly in your project or build your own library dependency out of it, whichever better suits your needs. There is also a very simple console application that shows how to evaluate JavaScript source code, and convert values from the JavaScript runtime into C# strings.

Building Apps with ChakraCore

There are a few extra steps involved to build C# applications with ChakraCore embedded. As of the time of writing, there are no public binaries for ChakraCore. But fret you must not, building ChakraCore is as easy as:

1. Cloning the Git repository from https://github.com/Microsoft/ChakraCore
2. Opening the solution https://github.com/Microsoft/ChakraCore/blob/master/Build/Chakra.Core.sln in Visual Studio (VS 2015 and the Windows 10 SDK are required if you wish to build for ARM).
3. Build the solution from Visual Studio.
4. The build output will be placed in BuildVcBuildbin relative to your Git root folder.

   If you wish to build from the command line, open up a Developer Command Prompt for Visual Studio, navigate to the Git root folder for ChakraCore and run:

   ```
   msbuild BuildChakra.Core.sln /p:Configuration=Debug /p:Platform=x86
   ```

   You'll want to replace the Configuration and Platform parameters with the proper settings for your build.

   Now that you have a version of ChakraCore.dll, you have some options for how to ship it with your application. The simplest way is to simply copy and paste the binary into your build output folder. For convenience, I drafted a simple MSBuild target to include in your .csproj to automatically copy these binaries for you each time you build:

   ```
   <Target Name="AfterBuild">
    <ItemGroup>
     <ChakraDependencies Include="$(ReferencesPath)ChakraCore.*" />
    </ItemGroup>
    <Copy SourceFiles="@(ChakraDependencies)" DestinationFolder="$(OutputPath) " />
   </Target>
   ```

   For those who don't speak MSBuild, one of the MSBuild conventions is to run targets in your

project named AfterBuild after the build is complete. The above bit of XML roughly translates to "after the build is complete, search the references path for files that match the pattern ChakraCore.* and copy those files to the output directory." You'll need to set the $(ReferencesPath) property in your .csproj as well.

If you are building your application for multiple platforms, then it helps to drop the ChakraCore.dll dependencies in folder names based off of your build configuration and platform. E.g., consider the following structure:

```
├── References
   ├── x86
      ├── Debug
         ├── ChakraCore.dll
         ├── ChakraCore.pdb
      ├── Release
         ├── ...
   ├── x64
      ├── ...
   ├── ARM
      ├── ...
```

That way you can declare the MSBuild property $(ReferencesPath) based off your build properties, e.g.

```
<ReferencesPath>References$(Configuration)$(Platform)</ReferencesPath>
```

ChakraCore JavaScript Data Marshalling

JavaScript Value Types

The first step to building more complex applications with ChakraCore is understanding the data model. JavaScript is a dynamic, untyped language that supports first-class functions. The data model for JavaScript values in ChakraCore obviously supports these designs. Here are the value types supported in Chakra:

- Undefined
- Null
- Number
- String
- Boolean
- Object
- Function
- Error
- Array

String Conversion With Serialization and Parsing

There are a number of ways of marshalling data from the CLR to the JavaScript runtime. A simple way is to parse and serialize the data as a JSON string once it enters the runtime, as follows:

```
var jsonObject = JavaScriptValue.GlobalObject.GetProperty(
   JavaScriptPropertyId.FromString("JSON"));
var stringify = jsonObject.GetProperty(
   JavaScriptPropertyId.FromString("stringify"))
var parse = jsonObject.GetProperty(
   JavaScriptPropertyId.FromString("parse"));
var jsonInput = @"{""foo"":42}";
var stringInput = JavaScriptValue.FromString(jsonInput);
var parsedInput = parse.CallFunction(JavaScriptValue.GlobalObject, stringInput);
var stringOutput = stringify.CallFunction(JavaScriptValue.GlobalObject, parsedInput);
var jsonOutput = stringOutput.ToString();
```

```
Debug.Assert(jsonInput == jsonOutput);
```

In the above code, we marshal the JSON data, {"foo":42} into the runtime as a string and parse the data using the JSON.parse function. The result is a JavaScript object, which we use as input to the JSON.stringify function, then use the ToString() method on the result value to put the result back into a .NET string. Obviously, the idea would be to use the parsedInput object as an input to your logic running in Chakra, and apply the stringify function only when you need to marshal data back out.

Direct Object Model Conversion (With Json.NET)

An alternative approach to the string-based approach in the previous section would be to use the Chakra native APIs to construct the objects directly in the JavaScript runtime. While you can choose whatever JSON data model you desire for your C# application, we chose Json.NET due to it's popularity and performance characteristics.

The basic outcome we are looking for is a function from JavaScriptValue (the Chakra data model) to JToken (the Json.NET data model) and the inverse function from JToken to JavaScriptValue. Since JSON is a tree data structure, a recursive visitor is a good approach for implementing the converters.

Here is the logic for the visitor class that converts values from JavaScriptValue to JToken:

```
public sealed class JavaScriptValueToJTokenConverter
{
   private static readonly JToken s_true = new JValue(true);
   private static readonly JToken s_false = new JValue(false);
   private static readonly JToken s_null = JValue.CreateNull();
   private static readonly JToken s_undefined = JValue.CreateUndefined();
private static readonly JavaScriptValueToJTokenConverter s_instance =
      new JavaScriptValueToJTokenConverter();
private JavaScriptValueToJTokenConverter() { }
public static JToken Convert(JavaScriptValue value)
   {
      return s_instance.Visit(value);
   }
private JToken Visit(JavaScriptValue value)
   {
      switch (value.ValueType)
      {
         case JavaScriptValueType.Array:
            return VisitArray(value);
         case JavaScriptValueType.Boolean:
            return VisitBoolean(value);
         case JavaScriptValueType.Null:
            return VisitNull(value);
         case JavaScriptValueType.Number:
            return VisitNumber(value);
         case JavaScriptValueType.Object:
            return VisitObject(value);
         case JavaScriptValueType.String:
            return VisitString(value);
         case JavaScriptValueType.Undefined:
            return VisitUndefined(value);
         case JavaScriptValueType.Function:
         case JavaScriptValueType.Error:
         default:
            throw new NotSupportedException();
      }
   }
private JToken VisitArray(JavaScriptValue value)
```

```
    {
        var array = new JArray();
        var propertyId = JavaScriptPropertyId.FromString("length");
        var length = (int)value.GetProperty(propertyId).ToDouble();
        for (var i = 0; i < length; ++i)
        {
            var index = JavaScriptValue.FromInt32(i);
            var element = value.GetIndexedProperty(index);
            array.Add(Visit(element));
        }
return array;
    }
private JToken VisitBoolean(JavaScriptValue value)
    {
        return value.ToBoolean() ? s_true : s_false;
    }
private JToken VisitNull(JavaScriptValue value)
    {
        return s_null;
    }
private JToken VisitNumber(JavaScriptValue value)
    {
        var number = value.ToDouble();
return number % 1 == 0
            ? new JValue((long)number)
            : new JValue(number);
    }
private JToken VisitObject(JavaScriptValue value)
    {
        var jsonObject = new JObject();
        var properties = Visit(value.GetOwnPropertyNames()).ToObject<string[]>();
        foreach (var property in properties)
        {
            var propertyId = JavaScriptPropertyId.FromString(property);
            var propertyValue = value.GetProperty(propertyId);
            jsonObject.Add(property, Visit(propertyValue));
        }
return jsonObject;
    }
private JToken VisitString(JavaScriptValue value)
    {
        return JValue.CreateString(value.ToString());
    }
private JToken VisitUndefined(JavaScriptValue value)
    {
        return s_undefined;
    }
}
```

And here is the inverse logic from JToken to JavaScript value:

```
public sealed class JTokenToJavaScriptValueConverter
{
    private static readonly JTokenToJavaScriptValueConverter s_instance =
        new JTokenToJavaScriptValueConverter();
private JTokenToJavaScriptValueConverter() { }
public static JavaScriptValue Convert(JToken token)
    {
        return s_instance.Visit(token);
```

```
  }
private JavaScriptValue Visit(JToken token)
  {
    if (token == null)
      throw new ArgumentNullException(nameof(token));
switch (token.Type)
    {
      case JTokenType.Array:
        return VisitArray((JArray)token);
      case JTokenType.Boolean:
        return VisitBoolean((JValue)token);
      case JTokenType.Float:
        return VisitFloat((JValue)token);
      case JTokenType.Integer:
        return VisitInteger((JValue)token);
      case JTokenType.Null:
        return VisitNull(token);
      case JTokenType.Object:
        return VisitObject((JObject)token);
      case JTokenType.String:
        return VisitString((JValue)token);
      case JTokenType.Undefined:
        return VisitUndefined(token);
      default:
        throw new NotSupportedException();
    }
  }
private JavaScriptValue VisitArray(JArray token)
  {
    var n = token.Count;
    var array = AddRef(JavaScriptValue.CreateArray((uint)n));
    for (var i = 0; i < n; ++i)
    {
      var value = Visit(token[i]);
      array.SetIndexedProperty(JavaScriptValue.FromInt32(i), value);
      value.Release();
    }
return array;
  }
private JavaScriptValue VisitBoolean(JValue token)
  {
    return token.Value<bool>()
      ? JavaScriptValue.True
      : JavaScriptValue.False;
  }
private JavaScriptValue VisitFloat(JValue token)
  {
    return AddRef(JavaScriptValue.FromDouble(token.Value<double>()));
  }
private JavaScriptValue VisitInteger(JValue token)
  {
    return AddRef(JavaScriptValue.FromDouble(token.Value<double>()));
  }
private JavaScriptValue VisitNull(JToken token)
  {
    return JavaScriptValue.Null;
  }
```

```
private JavaScriptValue VisitObject(JObject token)
    {
        var jsonObject = AddRef(JavaScriptValue.CreateObject());
        foreach (var entry in token)
        {
            var value = Visit(entry.Value);
            var propertyId = JavaScriptPropertyId.FromString(entry.Key);
            jsonObject.SetProperty(propertyId, value, true);
            value.Release();
        }
return jsonObject;
    }
private JavaScriptValue VisitString(JValue token)
    {
        return AddRef(JavaScriptValue.FromString(token.Value<string>()));
    }
private JavaScriptValue VisitUndefined(JToken token)
    {
        return JavaScriptValue.Undefined;
    }
private JavaScriptValue AddRef(JavaScriptValue value)
    {
        value.AddRef();
        return value;
    }
}
```

As with any recursive algorithm, there are base cases and recursion steps. In this case the base cases are the “leaf nodes” of the JSON tree (i.e., undefined, null, numbers, Booleans, strings) and the recursive steps occur when we encounter arrays and objects.

The goal of direct object model conversion is to lessen pressure on the garbage collector as serialization and parsing will generate a lot of intermediate strings. Bear in mind that your choice of .NET object model for JSON (Json.NET in the examples above) may also have an impact on your decision to use the direct object model conversion method outlined in this section or the string serialization / parsing method outlined in the previous section. If your decision is based purely on throughput, and your application is not GC-bound, the string-marshaling approach will outperform the direct object model conversion (especially with the back-and-forth overhead from native to managed code for large JSON trees).

You should evaluate the performance impact of either approach on your scenario before choosing one or the other. To assist in that investigation, I’ve published a simple tool for calculating throughput and garbage collection impact for both the CLR and Chakra on GitHub.

ChakraCore Threading Requirements

The ChakraCore runtime is single-threaded in the sense that only one thread may have access to it at a time. This does not mean, however, that you must designate a thread to do all the work on the JavaScriptRuntime (although it may be easier to do so).

Setting up the JavaScript runtime is relatively straightforward:

```
var runtime = JavaScriptRuntime.Create();
```

Before you can use this runtime on any thread, you must first set the context for a particular thread:

```
var context = runtime.CreateContext();
JavaScriptContext.Current = context;
```

When you are done using that thread for JavaScript work for the time being, be sure to reset the JavaScript context to an invalid value:

```
JavaScriptContext.Current = JavaScriptContext.Invalid;
```

At some later point, on any other thread, simply recreate or reassign the context as above. If you attempt to assign the context simultaneously on two different threads for the same runtime, ChakraCore will throw an exception. E.g.,

```
var t1 = Task.Run(() =>
{
   JavaScriptContext.Current = runtime.CreateContext();
   Task.Delay(1000).Wait();
   JavaScriptContext.Current = JavaScriptContext.Invalid;
});
var t2 = Task.Run(() =>
{
   JavaScriptContext.Current = runtime.CreateContext();
   Task.Delay(1000).Wait();
   JavaScriptContext.Current = JavaScriptContext.Invalid;
});
Task.WaitAll(t1, t2);
```

Is apt to throw an exception. Nothing should prevent you, however, from using multiple threads concurrently for two different runtimes.

Similarly, if you attempt to dispose the runtime without first resetting the context to an invalid value, ChakraCore will throw an exception notifying that the runtime is in use:

```
using (var runtime = JavaScriptRuntime.Create())
{
   var context = runtime.CreateContext();
   JavaScriptContext.Current = context;
}
```

If you encounter the “runtime is in use” exception that stems from disposing the runtime before unsetting the context, double check your JavaScript thread activity for any asynchronous behavior. The way async/await works in C# generally allows for any thread from the thread pool to carry out a continuation after the completion of an asynchronous operation. For ChakraCore to work properly, the context must be unset by the exact same physical thread (not logical thread) that set it initially. For more information, consult [MSDN https://msdn.microsoft.com/en-us/library/dd537609.aspx].

Thread Queue Options

In our implementation of the React Native on Windows, we considered a few different approaches to ensuring all JavaScript operations were single threaded. React Native has 3 main threads of activity, the UI thread, the background native module thread, and the JavaScript thread. Since JavaScript work can originate from either the native module thread or the UI thread, and generally speaking each thread does not block waiting for completion of activity on any other thread, we also have the requirement of implementing a FIFO queue for the JavaScript work.

ThreadPool Thread Capture

One of the options we considered was to block a thread pool thread permanently for evaluating JavaScript operations. Here’s the sample code for that:

```
// Initializes the thread queue
var queue = new BlockingCollection<Action>();
var asyncAction = ThreadPool.RunAsync(
   _ =>
   {
      JavaScriptContext.Current = context;
while (true)
      {
         var action = queue.Take();
         if (... /* Check disposal */) break;
try { action(); }
```

```
        catch (Exception ex) { ... /* Handle exceptions */ }
      }
JavaScriptContext.Current = JavaScriptContext.Invalid;
   },
   WorkItemPriority.Normal);
// Enqueues work
queue.Add(() => JavaScriptContext.RunScript(... /* JavaScript */);
```

The benefit to this approach is it's simplicity in that we know a single thread is running all of the JavaScript operations. The detriment is that we're permanently blocking a thread pool thread, so it cannot be used for other work.

Task Scheduler

Another approach we considered uses the .NET framework's TaskScheduler. There are a few ways to create a task scheduler that limits concurrency and guarantees FIFO, but for simplicity, we use this one from MSDN.

```
// Initializes the thread queue
var taskScheduler =
    new LimitedConcurrencyLevelTaskScheduler(1);
var taskFactory = new TaskFactory(taskScheduler);
// Enqueues work
taskFactory.StartNew(() =>
{
   if (... /* Check disposed */) return;
   try { JavaScriptContext.RunScript(... /* JavaScript */); }
   catch (Exception ex) { ... /* Handle exception */}
});
```

The benefit to this approach is that it does not require any blocking operations.

ChakraCore Runtime Considerations

Garbage Collection

One of the main deterrents from using ChakraCore in conjunction with another managed language like C# is the complexity of competing garbage collectors. ChakraCore has a few hooks to give you more control over how garbage collection in the JavaScript runtime is managed. For more information, checkout the documentation on runtime resource usage.

To JIT or not to JIT?

The option to run the just-in-time (JIT) compiler in ChakraCore is also completely optional. Obviously, without the JIT compiler, all the JavaScript code you run is fully interpreted. Depending on your application, you may want to weigh the overhead of the JIT compiler against the frequency at which you run certain functions. In case you do decide that the overhead of the JIT compiler is not worth the trade-off, here's how you can disable it:

```
var runtime =
   JavaScriptRuntime.Create(JavaScriptRuntimeAttributes.DisableNativeCodeGeneration);
```

From <https://www.microsoft.com/developerblog/2016/06/02/hybrid-apps-using-c-and-javascript-with-chakracore/>